

# **Principal Types for Object-Oriented Languages**

Andreas V. Hense

Gert Smolka

Fachbereich 14

Universität des Saarlandes

Technischer Bericht Nr. A 02/93

# Principal Types for Object-Oriented Languages

Andreas V. Hense<sup>†</sup> and Gert Smolka<sup>‡</sup>

June 23, 1993

## Abstract

Object-oriented languages can be translated into a  $\lambda$ -calculus with records. Therefore, type inference for record languages is one aspect of the yet unsolved problem of inferring types for object-oriented languages. In order to obtain the necessary flexibility for such a type system, we can either introduce a general subtyping notion or use extensible record types. Subtyping, especially in combination with imperative features, poses many hard problems. Therefore, the second approach is promising. The problem is that, in previous type inference systems that used extensible record types, principal types could not be inferred.

We have found that, for an object-oriented language where classes are not first-class citizens, we could greatly simplify the underlying record language. We show that, for our simple record language, there exists a type inference algorithm that infers principal types.

<sup>†</sup>Universität des Saarlandes, Im Stadtwald 15, 6600 Saarbrücken 11, Germany, *Tel.:* 0681/302-2464, *FAX:* 0681/302-4421, *e-mail:* [hense@cs.uni-sb.de](mailto:hense@cs.uni-sb.de)

<sup>‡</sup>Universität des Saarlandes and Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Stuhlsatzenhausweg 3, 6600 Saarbrücken 11, Germany

# 1 Introduction

Languages with records and subtypes are currently receiving much attention [1, 2, 5, 3, 6, 14, 20]. This intensive research activity is motivated by the endeavor to increase the flexibility and reliability of programming languages. Objects of object-oriented languages can be modeled as records. Subtyping, in those languages, roughly amounts to saying that an object (= record) having more methods (= components) than necessary can always “do the job”. Closely linked to the notion of ‘object’ is an internal state. It is however problematic to mix a general subtyping notion with imperative features [5]. This may have been the incentive behind Rémy’s [17] and Wand’s [25, 26] “subtyping without subtypes” by extensible record types. A variable which requires an  $a$ -component will have a type that is specific about this  $a$ -component, but is extensible in the sense that an object having the required  $a$ -component and further components also fits the type. For example, when typing the expression  $x.a$ , where  $x$  is a variable,  $a$  is a label, and the dot denotes record selection, we get  $\langle \alpha \mid a : \beta \rangle$  as an extensible record type for  $x$ . This means that  $x$  must be a record with an  $a$ -component and perhaps further components.

Extensible record types are quite subtle. Wand [21] thought at first that he could infer principal types<sup>1</sup> for records. He corrected the mistake [23] and finally came out with sets of principal types [25]. We felt that, for O’SMALL, the full generality of Wand’s language was not necessary. We considered a restricted language that contained fieldwise record adjunction instead of concatenation and thought that this would give us principal types [11]. There was, however, a subtle bug in the unification algorithm. Indeed, the language we need is even simpler. Since classes in O’SMALL are not first-class citizens, the compiler can resolve the record concatenations appearing in the modeling of class inheritance. The underlying language becomes surprisingly simple, and we are able to infer principal types for it. Principal types have the advantage over sets of principal types of being a much more concise documentation.

The paper is organized as follows. We present the record language **R** and its semantics by first-order rewrite rules. Types for this language are specified by an order-sorted signature. Readers unfamiliar with order-sorted logic will find basic definitions in [19]. Type inference rules are defined. It is shown that a term for which we can derive a type cannot go wrong. A type reconstruction algorithm infers principal types for **R**. By a two-phase formalization of the algorithm, we try to alleviate the problem of verification.

## 2 Expressions

For the definition of the abstract syntax of our simple record language **R**, we introduce some notation. Records are finite mappings and, therefore, sequences like  $a_1, \dots, a_n$  or  $a_1 : e_1, \dots, a_n : e_n$  or  $\alpha_1 \doteq \beta_1, \dots, \alpha_n \doteq \beta_n$  appear frequently. For our and, hopefully, the reader’s convenience, we abbreviate these terms to  $\bar{a}$ ,  $\bar{a}:\bar{e}$ , or  $\bar{\alpha}=\bar{\beta}$  respectively.

---

<sup>1</sup>Principal types are also called “most general types”.

Sometimes, a non-negative integer  $n$  or  $m$  will be referred to in the context of our abbreviating notation for sequences, although it is not made explicit in the notation.

Records can be regarded as finite mappings from labels to some kind of entries. The set of *labels* is countably infinite and totally ordered ( $\leq$ ). Labels are denoted by lower case letters  $a, b$  or  $a_1, a_2$  etc. A record that maps the labels  $a_1, \dots, a_n$  to the entries  $e_1, \dots, e_n$  is denoted as  $\langle \bar{a} \mapsto \bar{e} \rangle$ . In this notation, the labels  $a_1, \dots, a_n$  are always distinct.  $n$  is not mentioned explicitly and we assume that  $n \geq 0$ . Therefore, this notation includes the empty record. The empty record is also denoted as  $\langle \rangle$ . Variables are denoted by lower case letters  $x, y, z$ . Record selection is denoted by a dot.

**Definition 2.1 (Syntax of R)** The language **R** is defined by the following abstract syntax.

$e ::=$	$x$	variable
	$\langle \bar{a} \mapsto \bar{e} \rangle$	record
	$e.a$	selection

Variables are introduced in order to make the type inference system more interesting and to prepare for extensions of the language. The order of labels in a record plays no role. A record possessing a  $b$ -field is denoted as  $\langle \bar{a} \mapsto \bar{e}, b \mapsto e' \rangle$ . From the distinctness of labels it follows that  $b \notin \{a_1, \dots, a_n\}$ . Similarly,  $\langle \bar{a} \mapsto \bar{e}, \bar{b} \mapsto \bar{e}' \rangle$  is a record possessing fields for the labels  $a_1, \dots, a_n, b_1, \dots, b_m$  and  $\{\bar{a}\} \cap \{\bar{b}\} = \emptyset$ .

**Definition 2.2 (R-rewrite rules)** The following rewrite rule scheme stands for a countably infinite number of rules.

$$\langle \bar{a} \mapsto \bar{e}, b \mapsto e' \rangle.b \longrightarrow e' \quad (1)$$

The rewrite rule scheme applies to all label sets  $\{\bar{a}\}$  and labels  $b$  provided that they fulfill the distinctness condition claimed above. If a record has a field for the selected label the entry is the result of the selection. Otherwise, the selection term remains and represents an error.

**Theorem 2.3** The rewrite system of definition 2.2 is terminating and confluent.

**Proof:** The *size* of an **R**-term is defined as follows.<sup>2</sup>

$$\begin{aligned} |x| &= 1 \\ |e.a| &= |e| + 1 \\ |\langle \bar{a} \mapsto \bar{e} \rangle| &= 1 + |\bar{e}| \end{aligned}$$

The rewrite rule scheme strictly decreases the size of the term. This implies termination. Local confluence is shown by the convergence of all critical pairs. Confluence follows from local confluence and termination. Since we have no critical pairs we have confluence.  $\square$

As a consequence of this theorem, every term has a unique normal form.

<sup>2</sup>Here, the abbreviating notation stands for a sum:  $|\bar{e}| = \sum_{i=1}^n |e_i|$ .

### 3 Types

The main task of a type inference system is to guarantee the absence of type errors for accepted programs. The only type error in our simple language **R** is the selection of a label in a record that does not have this label. When we evaluate a term using the rewrite system we eventually reach a normal form. If the normal form of a ground term contains a selection it represents an error element. We may, however, have selections on variables in normal forms of non-ground terms. Since there is no way of binding variables there is no way of evaluating selections on them.

Order-sorted logic is used for expressing certain properties or constraints concerned with labels. These constraints could also be expressed in ordinary predicate logic but order-sorted logic makes them part of the formalism. There are two kinds of record types:

**closed record types** are types for “ordinary” records. They express that a record has the labels  $a$ ,  $b$ , and  $c$  at the top level, and that is all. Sorts for closed record types contain the finite set of labels that record types have at top level.

**open record types** express that a term must be a record having certain labels – and maybe more. Additional labels are “contained” in a so-called *row variable*. Open record types have always disjoint labels on one level. A row variable has a sort expressing that it does not range over labels occurring in the rest of the open record type on its level. Sorts for open record types contain the finite set of labels that the open record type does not range over and the set that it definitely does range over.

**Warning!** The words ‘sort’ and ‘type’ are both used and have different meanings. The expressions of the type language are defined in the framework of order-sorted logic and, thus, every type of the signature **RT** has a sort. Expressions of the language **R** will be related to types. An expression related to a type is said to have that type.

**Definition 3.1 (Signature RT)** We define the signature **RT** of types. The sort of all types is called **t**. Type terms are ranged over by variables  $\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \eta, \iota$ . There are infinitely many variables of any sort. Open and closed sorts are for record types. Closed sorts are simply finite sets of labels denoted as  $A$ . Open sorts are pairs of finite sets of labels denoted as  $(A, B)$ , where  $A \cap B = \emptyset$ .  $A$  is the set of labels that the record type does not range over and  $B$  is the set of labels that the record type definitely does range over. In the following subsort declarations  $A$  and  $B$  are finite sets of labels.

$$\begin{aligned} &(\emptyset, \emptyset) < \mathbf{t} \\ &A < (B, C) \Leftrightarrow C \subseteq A, A \cap B = \emptyset \\ &(A, B) < (C, D) \Leftrightarrow A \supseteq C, B \supseteq D \end{aligned}$$

Type terms are built by the following constructors.

$$\begin{aligned} \langle \bar{a} : \cdot \rangle &: \bar{\mathbf{t}} \rightarrow \{\bar{a}\} \\ \langle \cdot \mid \bar{a} : \cdot \rangle &: (A, B) \times \bar{\mathbf{t}} \rightarrow (A \setminus \{\bar{a}\}, B \cup \{\bar{a}\}) \quad \text{if } \{\bar{a}\} \subseteq A \end{aligned}$$

The first constructor is for a closed record type. It takes  $n$  inputs of an arbitrary sort.<sup>3</sup> The resulting term has a closed record sort with labels  $a_1, \dots, a_n$ . As already mentioned, the labels  $a_1, \dots, a_n$  are distinct.  $\{\bar{a}\}$  denotes the set  $\{a_1, \dots, a_n\}$ . The second constructor creates an open record type. On the left-hand side of the vertical bar, there is a term of an open record sort. The open record sort must exclude labels  $\bar{a}$ . The result is a record type having an open record sort excluding labels  $A \setminus \{\bar{a}\}$  and definitely having labels  $B \cup \{\bar{a}\}$ .

**Proposition 3.2** The signature **RT** is regular.

It is sometimes interesting to know whether two type terms have a common subsort. Sorts have a top element  $\mathbf{t}$ . We will add an artificial bottom element. This bottom element is the *empty sort* or the *inconsistent sort*. We call the other sorts *consistent*.

**Definition 3.3** Let  $S$  be the set of sorts of definition 3.1. Let  $\perp$  denote the inconsistent sort. Extend the subsort declaration of definition 3.1 by

$$\forall s \in S (\perp < s).$$

We denote the least partial order on  $S \cup \{\perp\}$  induced by this declaration and the subsort declarations of definition 3.1 by  $\leq$ . Overloading the symbol  $<$ , we write  $a < b$  for  $a \leq b$  and  $a \neq b$ .

**Proposition 3.4**  $(S \cup \{\perp\}, \leq)$  is a lattice.

The next propositions show how to calculate the greatest lower bounds of two record sorts and when two sorts are incompatible.

**Proposition 3.5**

$$A \sqcap B = \perp \Leftrightarrow A \neq B \quad (2)$$

$$(A, B) \sqcap (A', B') = \perp \Leftrightarrow A \cap B' \neq \emptyset \vee A' \cap B \neq \emptyset \quad (3)$$

$$A \sqcap (B, C) = \perp \Leftrightarrow A \cap B \neq \emptyset \vee C \not\subseteq A \quad (4)$$

**Proposition 3.6**

$$(A, B) \sqcap (A', B') = \begin{cases} \perp & \text{if } A \cap B' \neq \emptyset \vee A' \cap B \neq \emptyset \\ (A \cup A', B \cup B') & \text{otherwise} \end{cases} \quad (5)$$

$$A \sqcap (B, C) = \begin{cases} \perp & \text{if } A \cap B \neq \emptyset \vee C \not\subseteq A \\ A & \text{otherwise} \end{cases} \quad (6)$$

---

<sup>3</sup> $n$  is the implicit variable of our abbreviating notation for sequences.

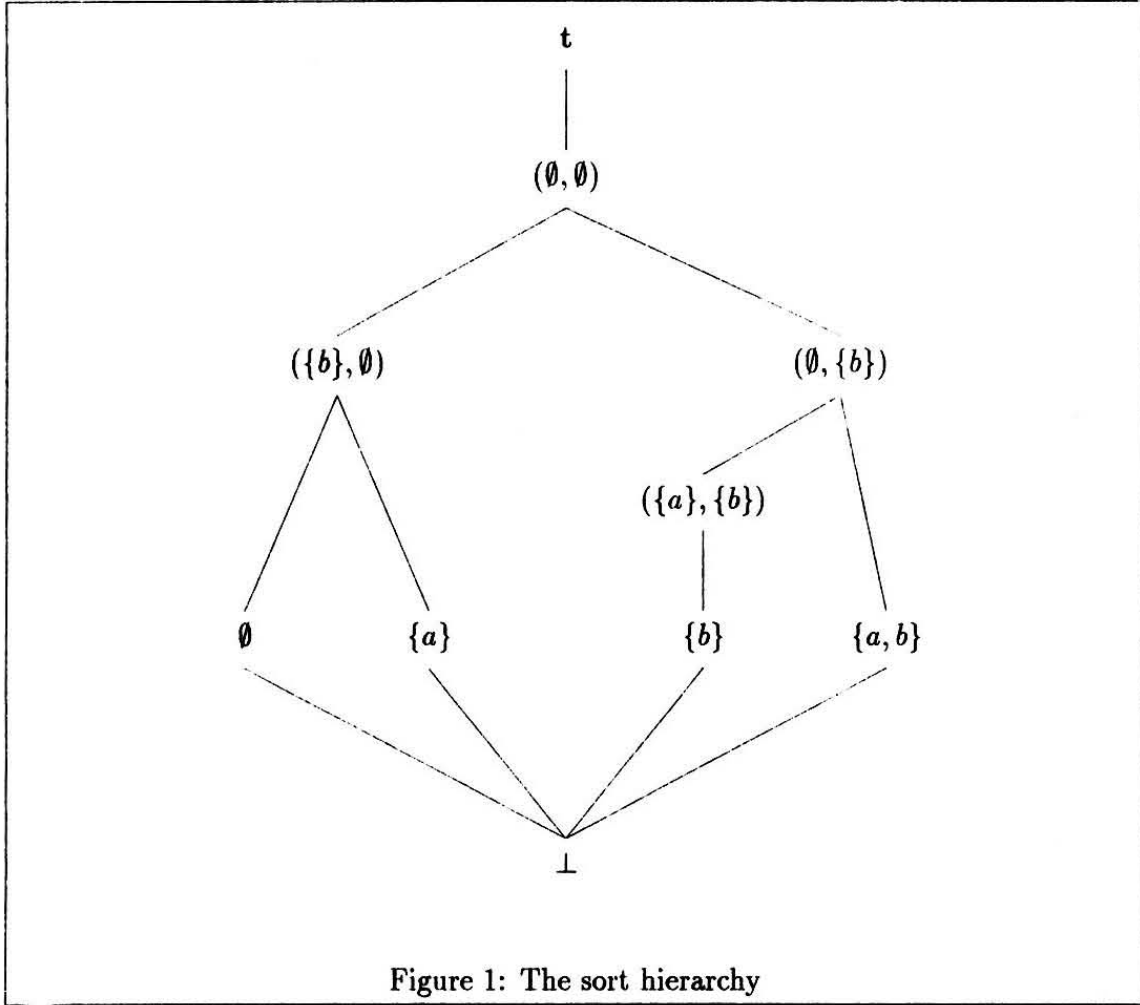


Fig. 1 shows part of the sort hierarchy. Just beneath the top element  $t$ , there is the greatest record sort  $(\emptyset, \emptyset)$ . The sort  $t$  has been introduced to allow for future extensions of the sorts, e.g., sorts for primitive types or function types. The sort  $(\emptyset, \emptyset)$  contains all record types because it does not exclude any labels, the left-hand side is empty, and it does not enforce any labels, the right-hand side is empty too. The open record sort  $(\{b\}, \emptyset)$  could be the sort of a row variable. Row variables, in general, exclude some labels but do not guarantee any.  $(\emptyset, \{b\})$  is an open record sort that enforces the label  $b$  but does not exclude any labels. Beneath it, there is the sort  $(\{a\}, \{b\})$  which excludes label  $a$  and enforces label  $b$ . This could be the sort of an open record type having a fixed  $b$ -component and a row variable of the sort  $(\{a, b\}, \emptyset)$ .

**Proposition 3.7**    The closed sorts are exactly the minimal consistent ones.

In many “ordinary” type systems, two types are different if they are syntactically different; e.g.  $\text{int} \rightarrow (\text{int} \rightarrow \text{int})$  is different from  $\text{int} \rightarrow \text{int}$ . In our type system, syntactically different type terms may denote the same type: e.g. the types  $\langle \rangle \mid \bar{a} : \bar{\alpha}$

and  $\langle \bar{a} : \bar{\alpha} \rangle$  are the same. Therefore, in definition 3.8, there are equations on type expressions – something which is a little unusual.

**Definition 3.8 (RT-equations)** Each of the following equations stands for a countably infinite number of equations.

$$\langle \langle \bar{a} : \bar{\alpha} \rangle \mid \bar{b} : \bar{\beta} \rangle \doteq \langle \bar{a} : \bar{\alpha}, \bar{b} : \bar{\beta} \rangle \quad (7)$$

$$\langle \langle \alpha \mid \bar{a} : \bar{\beta} \rangle \mid \bar{b} : \bar{\gamma} \rangle \doteq \langle \alpha \mid \bar{a} : \bar{\beta}, \bar{b} : \bar{\gamma} \rangle \quad (8)$$

All type terms used here are well-sorted. As a consequence, in equation (7), we have  $\{\bar{a}\} \cap \{\bar{b}\} = \emptyset$ . In equation (8),  $\alpha$  must have a sort excluding labels  $\{\bar{a}, \bar{b}\}$  and, again, we have  $\{\bar{a}\} \cap \{\bar{b}\} = \emptyset$ .

How are extensible record types used? Assume momentarily that we add integers, the type `int`, and  $\lambda$ -abstraction to our system. The function  $f := \lambda x. (x.a)$  can be applied to the record  $\langle a \mapsto 5 \rangle$ . Evaluating the application yields 5. It would be inflexible to infer the type  $\langle a : \alpha \rangle \rightarrow \alpha$  for  $f$  because we can apply  $f$  to  $\langle a \mapsto 5, b \mapsto 7 \rangle$ , which has type  $\langle a : \text{int}, b : \text{int} \rangle$ . Instead, we will infer an extensible record type for  $f$ , namely  $\langle \beta \mid a : \alpha \rangle \rightarrow \alpha$  where  $\beta$  is a *row variable*. A row variable is simply a type variable standing on the left-hand side of the bar in an open record type. The argument of the function  $f$  *must* have an  $a$ -field but *may* have more fields.

**Theorem 3.9** The rewrite system obtained by orienting the equations of definition 3.8 to the right is sort-decreasing, terminating, and confluent.

**Proof:** One easily verifies that orientation of the equations to the right yields rewrite rules (i.e.  $\sigma \rightarrow \tau$  is an **RT**-equation  $\sigma \doteq \tau$ ,  $\sigma$  is not a variable, and  $\mathcal{V}\tau \subset \mathcal{V}\sigma$ ) and that the rules are sort-decreasing. In order to show termination, we use the size of type terms. The *size* of an **RT**-term is defined as follows.

$$\begin{aligned} |\alpha| &= 1 \\ \langle \bar{a} : \bar{\sigma} \rangle &= 1 + |\bar{\sigma}| \\ \langle \sigma \mid \bar{a} : \bar{\tau} \rangle &= 1 + |\bar{\sigma}| + |\bar{\tau}| \end{aligned}$$

If a rewrite system is sort-decreasing, then it is locally confluent if and only if all critical pairs converge. If a relation is locally confluent and terminating then it is confluent. Thus, we can show confluence by the convergence of all critical pairs. We now list all overlaps with their corresponding substitutions and critical pairs:

Rules (8) and (7) overlap in the following way:

$$\begin{aligned} (\langle \langle \alpha \mid \bar{a} : \bar{\beta} \rangle \mid \bar{b} : \bar{\gamma} \rangle &\rightarrow \langle \alpha \mid \bar{a} : \bar{\beta}, \bar{b} : \bar{\gamma} \rangle, 1, \langle \langle \bar{c} : \bar{\delta} \rangle \mid \bar{a} : \bar{\epsilon} \rangle \rightarrow \langle \bar{c} : \bar{\delta}, \bar{a} : \bar{\epsilon} \rangle) \\ \theta = [\langle \bar{c} : \bar{\delta} \rangle / \alpha, \bar{\epsilon} / \bar{\beta}] &\quad (\langle \langle \bar{c} : \bar{\delta} \rangle \mid \bar{a} : \bar{\epsilon}, \bar{b} : \bar{\gamma} \rangle, \langle \langle \bar{c} : \bar{\delta}, \bar{a} : \bar{\epsilon} \rangle \mid \bar{b} : \bar{\gamma} \rangle) \end{aligned}$$

Rule (8) overlaps with itself in the following way:

$$\begin{aligned} (\langle \langle \alpha \mid \bar{a} : \bar{\beta} \rangle \mid \bar{b} : \bar{\gamma} \rangle &\rightarrow \langle \alpha \mid \bar{a} : \bar{\beta}, \bar{b} : \bar{\gamma} \rangle, 1, \langle \langle \delta \mid \bar{c} : \bar{\epsilon} \rangle \mid \bar{a} : \bar{\zeta} \rangle \rightarrow \langle \delta \mid \bar{c} : \bar{\epsilon}, \bar{a} : \bar{\zeta} \rangle) \\ \theta = [\langle \delta \mid \bar{c} : \bar{\epsilon} \rangle / \alpha, \bar{\zeta} / \bar{\beta}] &\quad (\langle \langle \delta \mid \bar{c} : \bar{\epsilon} \rangle \mid \bar{a} : \bar{\zeta}, \bar{b} : \bar{\gamma} \rangle, \langle \langle \delta \mid \bar{c} : \bar{\epsilon}, \bar{a} : \bar{\zeta} \rangle \mid \bar{b} : \bar{\gamma} \rangle) \end{aligned}$$

One easily verifies that the critical pairs converge.  $\square$



**Proposition 3.10** The sort of an **RT**-term in normal form is always determined by a tree of depth one.

In object-oriented languages, recursive types occur even in situations where one would not expect them in “ordinary” languages. The use of the pseudo variable **self** is the reason for this. We want to admit finitely representable infinite trees, called *rational trees* [16, 17] or *regular trees*. Therefore, the initial model is not sufficient and we have to explicitly state the axioms for our theory.

**Definition 3.11** The *theory of RT-terms* **RT** is given by equations (7) and (8) and the following axiom schemes.

$$\alpha \doteq \beta \rightarrow \perp \quad \{ \text{sort}(\alpha) \sqcap \text{sort}(\beta) = \perp \} \quad (9)$$

$$\begin{aligned} \langle \alpha \mid \bar{a} : \bar{\beta}, \bar{b} : \bar{\gamma} \rangle &\doteq \langle \delta \mid \bar{a} : \bar{\epsilon}, \bar{c} : \bar{\zeta} \rangle \\ &\rightarrow \exists \eta (\bar{\beta} \doteq \bar{\epsilon} \wedge \alpha \doteq \langle \eta \mid \bar{c} : \bar{\zeta} \rangle \\ &\quad \wedge \delta \doteq \langle \eta \mid \bar{b} : \bar{\gamma} \rangle) \quad \begin{cases} \{\bar{b}\} \cap \{\bar{c}\} = \emptyset \\ \text{sort}(\eta) = \{\bar{a}, \bar{b}, \bar{c}\} \end{cases} \end{aligned} \quad (10)$$

Axiom (9) states that types with incompatible sorts cannot be equal. Axiom (10) states that padding is possible if records have the proper row variables. The set of common labels is  $\{\bar{a}\}$ . The labels that differ are padded “crosswise” into the row variables  $\alpha$  and  $\delta$ . A new row variable  $\eta$  is introduced.

**Definition 3.12** We define a binary relation on types as

$$\sigma \approx \tau :\Leftrightarrow \sigma \models_{\mathbf{RT}} \tau$$

We call it the *congruence relation* because it is a congruence on every **RT**-algebra.

## 4 Typings

The type inference rules are formulated in the style of [9, 15]. A *sequent* is a triple  $\Gamma \vdash e : \tau$ . We read “term  $e$  has type  $\tau$  in the type environment  $\Gamma$ ”. Extending our abbreviating notation, we write  $\Gamma \vdash \bar{e} : \bar{\sigma}$  for  $\Gamma \vdash e_1 : \sigma_1 \dots \Gamma \vdash e_n : \sigma_n$ . A *type environment* is a finite mapping from term variables to types. It is written as  $[\bar{x} : \bar{\tau}]$ .

Let  $\Gamma$  be a type environment. In  $\Gamma \cdot [x : \tau]$ , the type environment  $\Gamma$  is either extended by an entry for  $x$  or an existing entry is overwritten. A type is retrieved from a type environment by  $\Gamma(x)$ . The inference rules are shown in Fig. 2. Side conditions for the applicability of rules are written on the right-hand side of an opening brace. With  $\mathcal{V}e$  we denote the *free variables* of the term  $e$ .  $\mathcal{D}f$  denotes the *domain* of a function.

$$\frac{}{\Gamma \vdash x : \tau} \{ \Gamma(x) = \tau \} \quad (\text{VAR})$$

$$\frac{\Gamma \vdash \bar{e} : \bar{\sigma}}{\Gamma \vdash \langle \bar{a} \mapsto \bar{e} \rangle : \langle \bar{a} : \bar{\sigma} \rangle} \quad (\text{REC})$$

$$\frac{\Gamma \vdash e : \langle \sigma \mid a : \tau \rangle}{\Gamma \vdash e.a : \tau} \quad (\text{SEL})$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \tau} \{ \sigma \approx \tau \} \quad (\text{EQ})$$

Figure 2: Type inference rules for records

**Definition 4.1** Let  $e$  be a term and  $\Gamma$  a type environment with  $\mathcal{D}\Gamma = \mathcal{V}e$ . If, for some  $\sigma$ , we have  $\Gamma \vdash e : \sigma$  then we say that  $(\Gamma, \sigma)$  is a *typing* of  $e$ . We call a term  $e$  *well-typed* if it has a typing.

**Definition 4.2** A *type substitution* maps type variables to types. Type substitutions are ranged over by  $\theta$  and  $\psi$ . We write the application of a type substitution  $\theta$  to a type  $\sigma$  as the juxtaposition  $\theta\sigma$ . We write  $\theta\Gamma$  for the application of  $\theta$  to every component of the type environment  $\Gamma$ .

**Definition 4.3** Let  $\mathcal{D}\Gamma = \mathcal{D}\Gamma'$ . The typing  $(\Gamma, \sigma)$  is *more general than* the typing  $(\Gamma', \sigma')$  if and only if there exists a type substitution  $\theta$  such that  $\theta\sigma \approx \sigma'$  and, for all  $x \in \mathcal{D}\Gamma$ ,  $\theta(\Gamma(x)) \approx \Gamma'(x)$ .

**Proposition 4.4** The relation ‘more general than’ is a preorder on typings.

**Definition 4.5 (Principal typing)** We call a typing  $(\Gamma, \tau)$  of  $e$  *principal* if and only if it is more general than all other typings of  $e$ .

**Lemma 4.6 (Rewriting preserves types)** If  $(\Gamma, \sigma)$  is a typing of an **R**-term  $e$  and  $e \rightarrow e'$ , then  $(\Gamma, \sigma)$  is a typing of  $e'$ .

**Proof:** It suffices to show that we can infer the same type for the right-hand side of the rewrite rule as for the left-hand side. For the left-hand side, we have the proof tree

$$\frac{\frac{\Gamma \vdash \bar{e} : \bar{\sigma} \quad \Gamma \vdash e' : \tau}{\Gamma \vdash \langle \bar{a} \mapsto \bar{e}, b \mapsto e' \rangle : \langle \bar{a} : \bar{\sigma}, b : \tau \rangle} \quad (\text{REC})}{\Gamma \vdash \langle \bar{a} \mapsto \bar{e}, b \mapsto e' \rangle : \langle \langle \bar{a} : \bar{\sigma} \rangle \mid b : \tau \rangle} \quad (\text{EQ})$$

$$\frac{}{\Gamma \vdash \langle \bar{a} \mapsto \bar{e}, b \mapsto e' \rangle.b : \tau} \quad (\text{SEL})$$

One of the premises of this proof tree is  $\Gamma \vdash e' : \tau$ . This would be the root of the proof tree for the right-hand side of the rewrite rule.  $\square$

**Theorem 4.7 (Well-typed terms do not go wrong)** Any well-typed ground term reduces to a normal form that does not contain any selections.

**Proof:** By lemma 4.6, a normal form of a well-typed term is also well-typed. From the type inference rules for **R**, we see that every subterm of a well-typed term is also well-typed. Choose any innermost selection subterm, i.e. a term  $e.b$  where  $e$  does not contain any selection. Then  $e$  must be of the form  $\langle \bar{a} \mapsto \bar{e} \rangle$ . There are two cases:

$b \in \{\bar{a}\}$  In this case, we can apply rule (1). This contradicts our assumption that the term is in normal form.

$b \notin \{\bar{a}\}$  In this case, we cannot infer a type for our subterm. This contradicts our assumption that the whole term is well-typed.  $\square$

## 5 Type Reconstruction

The algorithm that finds a typing for a term consists of two orthogonal phases. The first phase creates constraints as it decomposes the input term. The second phase solves the constraints.

The algorithm works on a structure called “frame”. A frame consists of a type environment, proof obligations of the form  $e : \tau$ , and equational constraints.

**Definition 5.1** A *frame* is a quadruple written as  $\exists \bar{\alpha}(\Gamma; \mu; \phi)$ , where

- $\exists \bar{\alpha}$  is the existential quantification of type variables,
- $\Gamma$  is a type environment,
- $\mu$  is a conjunction of proof obligations  $e : \alpha$ , where  $e$  is a term and  $\alpha$  a type variable,
- $\phi$  is a conjunction of equational constraints  $\alpha \doteq \tau$ , where  $\alpha$  is a type variable and  $\tau$  a type.

Frames have semantics. Intuitively, a frame has a solution if all proof obligations can be fulfilled using the type environment. The proof obligations are only of the form  $e : \alpha$ , where  $\alpha$  is a type variable. The types themselves are encoded in the equational constraints. Thus, we can also represent recursive types.

**Definition 5.2** A mapping  $\theta$  is a *solution of a frame*  $\exists \bar{\alpha}(\Gamma; \bar{e}:\bar{\sigma}; \phi)$  if and only if there is a type substitution  $\psi$  such that  $\theta$  and  $\psi$  agree everywhere except possibly on  $\bar{\alpha}$  and

1.  $\psi\Gamma \vdash \bar{e} : \psi\bar{\sigma}$ ,

2. for all  $\alpha \doteq \tau \in \phi$  :  $\psi\alpha \approx \psi\tau$ .

We denote the set of all solutions of the frame  $\exists\bar{\alpha}(\Gamma; \bar{e}:\bar{\sigma}; \phi)$  by  $\llbracket \exists\bar{\alpha}(\Gamma; \bar{e}:\bar{\sigma}; \phi) \rrbracket$ .

The mapping  $\psi$  in the above definition expresses the usual interpretation [12] of the existential quantification  $\exists\bar{\alpha}$  in the frame.

## 5.1 Constraint Extraction

The first phase creates constraints as it decomposes the input term. When the rules of the next definition are applied to a frame, its middle component is consumed while its rightmost component increases.

**Definition 5.3** We define *frame simplification rules* for the language **R**.

$$\frac{\exists\bar{\alpha}(\Gamma; x : \alpha \wedge \mu; \phi)}{\exists\bar{\alpha}(\Gamma; \mu; \phi \wedge \alpha \doteq \tau)} \left\{ \Gamma(x) = \tau \right. \quad (11)$$

$$\frac{\exists\bar{\alpha}(\Gamma; \langle \bar{a} \mapsto \bar{e} \rangle : \alpha \wedge \mu; \phi)}{\exists\bar{\alpha}, \bar{\beta}(\Gamma; \bar{e} : \bar{\beta} \wedge \mu; \phi \wedge \alpha \doteq \langle \bar{a} : \bar{\beta} \rangle)} \left\{ \text{sort}(\bar{\beta}) = \bar{t} \right. \quad (12)$$

$$\frac{\exists\bar{\alpha}(\Gamma; e.a : \alpha \wedge \mu; \phi)}{\exists\bar{\alpha}, \beta, \gamma(\Gamma; e : \beta \wedge \mu; \phi \wedge \beta \doteq \langle \gamma \mid a : \alpha \rangle)} \left\{ \begin{array}{l} \text{sort}(\beta) = t \\ \text{sort}(\gamma) = (\{a\}, \emptyset) \end{array} \right. \quad (13)$$

In rule (11), the type of the variable  $x$  is retrieved from the type environment. In rule (12), the record is split up as one would expect. For each record field, we introduce a new type variable of the maximal sort. In rule (13), the newly introduced variable  $\gamma$  has the correct sort that guarantees the disjointness of labels in record types. The newly introduced variable  $\beta$  has the top sort.

**Lemma 5.4** The frame simplification rules leave the set of solutions invariant.

**Proof:** In this proof by case analysis on the frame simplification rules, we are using definition 5.2 extensively. Each time we show that “ $\theta$  is a solution of the frame on top of the rule if and only if it is a solution of the frame on the bottom of the rule”. Since we will be using a mapping  $\psi$  agreeing with  $\theta$  except possibly on the existentially quantified variables, we will not mention this fact and only talk about  $\psi$ . We will also not mention that  $\psi$  must fulfill  $\mu$  and  $\phi$  because  $\mu$  and  $\phi$  appear in every frame simplification rule on top and on the bottom. We abbreviate  $\theta \in \llbracket \dots \text{top frame} \dots \rrbracket$  to “top” and  $\theta \in \llbracket \dots \text{bottom frame} \dots \rrbracket$  to “bottom”.

(11)

$$\begin{array}{lcl} \text{top} & \Leftrightarrow & \psi\Gamma \cdot [x : \tau] \vdash x : \psi\alpha \\ & \stackrel{(\text{EQ})(\text{VAR})}{\Leftrightarrow} & \psi\tau \models_{\mathbf{R}} \psi\alpha \\ & \Leftrightarrow & \text{bottom} \end{array}$$

(12)

$$\begin{aligned}
\text{top} &\Leftrightarrow \psi\Gamma \vdash \langle \bar{a} \mapsto \bar{c} \rangle : \psi\alpha \\
&\stackrel{(\text{EQ})}{\Leftrightarrow} \stackrel{(\text{REC})}{\Leftrightarrow} \psi\Gamma \vdash \bar{c} : \psi\bar{\beta}, \quad \psi\alpha \models_{\mathbf{RT}} \psi\langle \bar{a} : \bar{\beta} \rangle \\
&\Leftrightarrow \text{bottom}
\end{aligned}$$

(13)

$$\begin{aligned}
\text{top} &\Leftrightarrow \psi\Gamma \vdash e.a : \psi\alpha \\
&\stackrel{(\text{EQ})}{\Leftrightarrow} \stackrel{(\text{SEL})}{\Leftrightarrow} \psi\Gamma \vdash e : \psi\langle \gamma \mid a : \alpha \rangle \\
&\Leftrightarrow \psi\Gamma \vdash e : \psi\beta \wedge \psi\beta \models_{\mathbf{RT}} \psi\langle \gamma \mid a : \alpha \rangle \\
&\Leftrightarrow \text{bottom}
\end{aligned}$$

□

## 5.2 Constraint Resolution

The second phase operates on the third component of a frame. It tries to resolve the constraints that were created by the first phase (definition 5.3). As already mentioned, we want to admit rational trees as solutions. The constraints created by the first phase are flat in the sense that they are of the form  $\alpha \doteq \beta$ ,  $\alpha \doteq \langle \bar{a} : \bar{\beta} \rangle$ , or  $\alpha \doteq \langle \beta \mid \bar{b} : \bar{\gamma} \rangle$ . There is at most one constructor on the right-hand side. The only constraints that are not flat are introduced by rule (11) because the types in the type environment are not restricted to flat types. However, we assume that the non-flat types are flattened by introducing new existentially quantified variables and new equations. This is always possible in first-order logic. The input is a conjunction of equations on types with existential quantors on the outside. Newly introduced variables are supposed to be existentially quantified at the outside too. Types are always kept in normal form during the resolution process. The algorithm consists in applying rules (14) through (25) to the constraint  $\phi$ . The order of application does not matter. For better readability, we write equations connected by  $\wedge$  one upon another and omit the  $\wedge$ -sign. The first three rules eliminate repeated occurrences of the same variable on the left-hand side.

$$\frac{\phi \quad \alpha \doteq \alpha}{\phi} \quad (14)$$

$$\frac{\phi \quad \alpha \doteq \beta}{\begin{array}{l} [\beta/\alpha]\phi \\ \alpha \doteq \beta \end{array}} \left\{ \begin{array}{l} \alpha \neq \beta \\ \text{sort}(\alpha) \geq \text{sort}(\beta) \\ \alpha \in \mathcal{V}\phi \end{array} \right. \quad (15)$$

$$\begin{array}{c}
\phi \\
\alpha \doteq \sigma \\
\alpha \doteq \tau \\
\hline
\perp
\end{array}
\left\{ \text{sort}(\sigma) \sqcap \text{sort}(\tau) = \perp \right.
\quad (16)$$

The side conditions of rule (15) are worth a comment. The sorts of the variables  $\alpha$  and  $\beta$  may be equal. Termination (theorem 5.6) is ensured by the last condition requiring that the variable  $\alpha$  is not isolated.

The next four rules eliminate repeated occurrences of the same variable on the left-hand side when record types are involved.

$$\begin{array}{c}
\phi \\
\alpha \doteq \langle \bar{a} : \bar{\beta} \rangle \\
\alpha \doteq \langle \bar{a} : \bar{\gamma} \rangle \\
\hline
\phi \\
\alpha \doteq \langle \bar{a} : \bar{\beta} \rangle \\
\bar{\beta} \doteq \bar{\gamma}
\end{array}
\quad (17)$$

$$\begin{array}{c}
\phi \\
\alpha \doteq \langle \bar{a} : \bar{\beta}, \bar{b} : \bar{\gamma} \rangle \\
\alpha \doteq \langle \delta \mid \bar{b} : \bar{\epsilon} \rangle \\
\hline
\phi \\
\alpha \doteq \langle \bar{a} : \bar{\beta}, \bar{b} : \bar{\gamma} \rangle \\
\delta \doteq \langle \bar{a} : \bar{\beta} \rangle \\
\bar{\gamma} \doteq \bar{\epsilon}
\end{array}
\quad (18)$$

$$\begin{array}{c}
\phi \\
\alpha \doteq \langle \beta \mid \bar{a} : \bar{\gamma} \rangle \\
\alpha \doteq \langle \delta \mid \bar{a} : \bar{\epsilon} \rangle \\
\hline
\phi \\
\alpha \doteq \langle \beta \mid \bar{a} : \bar{\gamma} \rangle \\
\beta \doteq \delta \\
\bar{\gamma} \doteq \bar{\epsilon}
\end{array}
\quad (19)$$

$$\begin{array}{c}
\phi \\
\alpha \doteq \langle \beta \mid \bar{a} : \bar{\gamma}, \bar{b} : \bar{\delta} \rangle \\
\alpha \doteq \langle \epsilon \mid \bar{a} : \bar{\zeta}, \bar{c} : \bar{\psi} \rangle \\
\hline
\phi \\
\alpha \doteq \langle \iota \mid \bar{a} : \bar{\gamma}, \bar{b} : \bar{\delta}, \bar{c} : \bar{\psi} \rangle \\
\beta \doteq \langle \iota \mid \bar{c} : \bar{\psi} \rangle \\
\epsilon \doteq \langle \iota \mid \bar{b} : \bar{\delta} \rangle \\
\bar{\gamma} \doteq \bar{\zeta}
\end{array}
\left\{ \begin{array}{l} \{\bar{b}\} \cup \{\bar{c}\} \neq \emptyset \\ \{\bar{b}\} \cap \{\bar{c}\} = \emptyset \end{array} \right.
\quad (20)$$

Rule (17) analyzes two closed record types. Closed record types can only be equal if their set of labels is equal. If the set of labels is not equal rule (16) can be applied. Rule (18) analyzes a closed and an open record type. The open record type must be padded in order to get the same label set. Rule (19) compares two open record types having the same set of explicit labels. Rule (20) compares two open record types having different sets of explicit labels. The missing labels are padded in “crosswise”. On the right of the brace, there are sometimes side conditions for the applicability of the rule.

The following four rules eliminate all equations  $\alpha \doteq \sigma$  for which  $\text{sort}(\alpha) \geq \text{sort}(\sigma)$  does not hold. The first three rules are of general nature, the fourth one is concerned with records.

$$\frac{\phi \quad \alpha \doteq \sigma}{\perp} \left\{ \text{sort}(\alpha) \sqcap \text{sort}(\sigma) = \perp \right. \quad (21)$$

$$\frac{\phi \quad \alpha \doteq \beta}{\phi} \left\{ \text{sort}(\alpha) < \text{sort}(\beta) \right. \quad (22)$$

$$\beta \doteq \alpha$$

$$\frac{\phi \quad \alpha \doteq \beta}{\phi} \left\{ \begin{array}{l} \text{sort}(\alpha) \sqcap \text{sort}(\beta) = s \\ s \neq \perp \\ s \neq \text{sort}(\alpha) \\ s \neq \text{sort}(\beta) \end{array} \right. \quad (23)$$

$$\frac{\phi \quad \alpha \doteq \gamma \quad \beta \doteq \gamma}{\phi} \left\{ \begin{array}{l} \text{sort}(\gamma) = s \end{array} \right.$$

$$\frac{\phi \quad \alpha \doteq \langle \beta \mid \bar{a} : \bar{\gamma} \rangle}{\phi} \left\{ \begin{array}{l} \text{sort}(\alpha) \sqcap \text{sort}(\langle \beta \mid \bar{a} : \bar{\gamma} \rangle) = s \\ s \neq \text{sort}(\langle \beta \mid \bar{a} : \bar{\gamma} \rangle) \\ s = \text{sort}(\langle \delta \mid \bar{a} : \bar{\gamma} \rangle) \end{array} \right. \quad (24)$$

$$\frac{\phi \quad \alpha \doteq \langle \delta \mid \bar{a} : \bar{\gamma} \rangle}{\phi} \quad \beta \doteq \delta$$

In rule (24), the sort of the new variable  $\delta$  must be chosen such that the side conditions are fulfilled. This rule can be applied if the explicit labels  $\bar{a}$  do not disagree with  $\alpha$  but the row variable  $\beta$  has a sort too big for  $\alpha$ . We then replace  $\beta$  by a row variable with a smaller sort. What is the sort of  $\delta$  in rule (24)? If we know that  $s = \text{sort}(\langle \delta \mid \bar{a} : \bar{\gamma} \rangle)$  and  $s$  is given, the sort of the row variable  $\delta$  is also fixed: If  $s = B \uplus \{\bar{a}\}$ , then  $\text{sort}(\delta) = B$ . If  $s = (B, \{\bar{a}\})$ , then  $\text{sort}(\delta) = B$ .

The last rule does not fit into any of the above classifications. It eliminates row

variables that are equal to the empty record type.

$$\begin{array}{c}
 \phi \\
 \alpha \doteq \langle \beta \mid \bar{a} : \bar{\gamma} \rangle \\
 \beta \doteq \langle \rangle \\
 \hline
 \phi \\
 \alpha \doteq \langle \bar{a} : \bar{\gamma} \rangle \\
 \beta \doteq \langle \rangle
 \end{array} \tag{25}$$

This concludes the constraint resolution rules. The ensemble of these rules represent an algorithm whose correctness we are going to prove now.

**Definition 5.5** A variable  $\alpha$  is called *isolated* in a conjunction of equations  $\phi$  if there is an equation  $\alpha \doteq \sigma$  in  $\phi$  and this is the only occurrence of  $\alpha$  in  $\phi$ .

**Theorem 5.6 (Termination)** There is no infinite chain of applications of rules (14) through (25).

**Proof:** In order to show the termination of the resolution step we define functions  $r_1$  through  $r_5$  that map the constraint  $\phi$  into well-founded domains. We define a size function on constraints as the lexicographical order on  $(r_1, r_2, r_3, r_4, r_5)$ , where

$$\begin{aligned}
 r_1 &= \{ \{ \text{sort}(\alpha) \mid \beta \doteq \langle \alpha \mid \bar{a} : \bar{\gamma} \rangle \text{ is an occurrence of an equation in } \phi \} \} \\
 r_2 &= \{ \{ |\sigma| \mid \alpha \doteq \sigma \text{ is an occurrence of an equation in } \phi \text{ and } \sigma \text{ is not a variable} \} \} \\
 r_3 &= \{ \{ \text{sort}(\alpha) \mid \beta \doteq \alpha \text{ is an occurrence of an equation in } \phi \} \} \\
 r_4 &= | \{ \{ \alpha \doteq \sigma \text{ is an occurrence of an equation in } \phi \} \} | \\
 r_5 &= \{ \alpha \mid \alpha \text{ occurs in } \phi, \alpha \text{ is not isolated} \}
 \end{aligned}$$

We will now show for each of the rules in question that their application to a constraint decreases its size. Since the order is lexicographic, it suffices to find one component with  $>$  where all previous components are  $\geq$ .

rule	$r_1$	$r_2$	$r_3$	$r_4$	$r_5$
14	=	=	>		
15	$\geq$	=	$\geq$	=	>
16	$\geq$	$\geq$	$\geq$	>	
17	=	>			
18	>				
19	>				
20	>				
21	$\geq$	$\geq$	$\geq$	>	
22	=	=	>		
23	=	=	>		
24	>				
25	>				



Termination follows from the existence of a minimal element in the codomains of the functions and the absence of infinite chains for multiset replacements (theorem by Dershowitz and Manna [10]). Sorts contain no infinite decreasing chains for fixed programs because the set of labels in a frame is finite.  $\square$

One may ask oneself why we bother to introduce a countably infinite number of labels while we must argue with finite label sets in frames for the termination of our resolution step. An infinite number of labels is needed for the incrementality of the algorithm. We want to infer the type of a program once and for all, no matter where this program is used. It is exactly due to the unknown uses of a program that the labels cannot be reduced to a finite number.

**Definition 5.7** A *solved form* of a conjunction of equations between types has the form  $\alpha_1 \doteq \sigma_1 \wedge \dots \wedge \alpha_n \doteq \sigma_n$  with distinct variables  $\alpha_1, \dots, \alpha_n$  and, for all  $1 \leq i \leq n$ , the sort of  $\sigma_i$  is a subsort of that of  $\alpha_i$ .

**Theorem 5.8 (Effectiveness)** After the application of rules (14) through (25)), we either have a solved form or failure.

**Proof:** We first show that variables on left-hand sides of equations are distinct. We assume that a variable occurs more than once on the left-hand side and show that this contradicts our assumption that the algorithm has already terminated.

$\alpha \doteq \beta \wedge \alpha \doteq \sigma$  We must have  $\alpha \neq \beta$  since, otherwise, rule (14) would be applicable. If  $\text{sort}(\alpha) \geq \text{sort}(\beta)$  then rule (15) is applicable. If  $\text{sort}(\alpha) < \text{sort}(\beta)$  then rule (22) is applicable. If a common subsort exists we can apply rule (23), otherwise, rule (21).

$\alpha \doteq \langle \bar{a} : \bar{\beta} \rangle \wedge \alpha \doteq \langle \bar{b} : \bar{\gamma} \rangle$  If  $\{\bar{a}\} = \{\bar{b}\}$  then rule (17) is applicable, otherwise, rule (16).

$\alpha \doteq \langle \bar{a} : \bar{\beta} \rangle \wedge \alpha \doteq \langle \delta \mid \bar{b} : \bar{\gamma} \rangle$  If  $\{\bar{a}\} \supseteq \{\bar{b}\}$  then rule (18) is applicable, otherwise, rule (16).

$\alpha \doteq \langle \beta \mid \bar{a} : \bar{\gamma} \rangle \wedge \alpha \doteq \langle \delta \mid \bar{b} : \bar{\epsilon} \rangle$  If  $\{\bar{a}\} = \{\bar{b}\}$  then rule (19) is applicable, otherwise, rule (20).

It remains to show that the sorts of the right-hand sides of equations are subsorts of the respective left-hand sides. In the following cases, we always assume that there remains an equation  $\alpha \doteq \sigma$  and that  $\text{sort}(\alpha) \geq \text{sort}(\sigma)$  does not hold. The results are contained in the following table. Above, we list the sort relation of  $\alpha$  and  $\sigma$ . On the left, we list the possible forms of  $\sigma$ . The table entries consist of the applicable rules or a comment that a combination is impossible.

	$\text{sort}(\alpha) < \text{sort}(\sigma)$	$\text{sort}(\alpha) \sqcap \text{sort}(\sigma) = s$ $s \neq \perp$ $s < \text{sort}(\alpha)$ $s < \text{sort}(\sigma)$	$\text{sort}(\alpha) \sqcap \text{sort}(\sigma) = \perp$
$\beta$	(14), or (22)	(23)	(21)
$\langle \bar{a} : \bar{\beta} \rangle$	impossible	impossible	(21)
$\langle \beta \mid \bar{b} : \bar{\gamma} \rangle$	(24), or (21)	(24), or (21)	(21)

□

**Lemma 5.9** The application of rules (14) through (25) leaves the set of solutions invariant.

**Proof:** We show invariance for any model  $\mathcal{A}$  of the theory RT by case analysis over the rules. Since the proofs for rules (14), (15), (22), (23), (24), and (25) are trivial we concentrate on the remaining ones.

(16)

$$\begin{aligned}
& (\phi \wedge \alpha \doteq \sigma \wedge \alpha \doteq \tau)^{\mathcal{A}} \\
& \subseteq \{s \mid [\alpha]_s = [\sigma]_s \wedge [\alpha]_s = [\tau]_s\} \\
& = \{s \mid [\alpha]_s = [\sigma]_s \wedge [\sigma]_s = [\tau]_s\} \\
& \subseteq \{s \mid [\sigma]_s = [\tau]_s\} \\
& \stackrel{(9)}{=} (\perp)^{\mathcal{A}}
\end{aligned}$$

(21) This case is similar to the previous one.

(17)

$$\begin{aligned}
& (\phi \wedge \alpha \doteq \langle \bar{a} : \bar{\beta} \rangle \wedge \alpha \doteq \langle \bar{a} : \bar{\gamma} \rangle)^{\mathcal{A}} \\
& = \{s \mid s \in (\phi)^{\mathcal{A}} \wedge [\alpha]_s = [\langle \bar{a} : \bar{\beta} \rangle]_s \wedge [\langle \bar{a} : \bar{\beta} \rangle]_s = [\langle \bar{a} : \bar{\gamma} \rangle]_s\} \\
& \stackrel{(10)}{=} \{s \mid s \in (\phi)^{\mathcal{A}} \wedge [\alpha]_s = [\langle \bar{a} : \bar{\beta} \rangle]_s \wedge [\bar{\beta}]_s = [\bar{\gamma}]_s\} \\
& = (\phi \wedge \alpha \doteq \langle \bar{a} : \bar{\beta} \rangle \wedge \bar{\beta} \doteq \bar{\gamma})^{\mathcal{A}}
\end{aligned}$$

(18), (19), (20) These cases are similar to the previous one.

□

**Theorem 5.10 (Principal types)** Let  $\mathcal{V}e = \{\bar{x}\}$ . If we start the algorithm with the frame  $\exists \bar{\beta}([\bar{x}:\bar{\beta}]; e : \alpha; T)$ , where  $\beta_1, \dots, \beta_n$ , and  $\alpha$  are disjoint, it computes a principal typing if  $e$  has a typing, or stops with failure.

**Proof:** The type environment in a typing of a term  $e$  must contain exactly the free variables of  $e$ . The solutions of  $\exists \bar{\beta}([\bar{x}:\bar{\beta}]; e : \alpha; T)$  contain all typings of  $e$ . Thus, if we can transform the frame into a solved form, i.e. a substitution, we have found a principal typing because both phases of the algorithm leave the set of solutions invariant (lemmata 5.4 and 5.9). □

We have seen that the whole algorithm consists of two orthogonal phases. The first phase creates constraints as it decomposes the input term. The second phase solves the constraints. The conceptually simplest way of proceeding is to complete the first phase before starting the second one. In practice, one is interested in localizing type errors. The first phase creates constraints and can never fail because it does not try to solve them. If the second phase were started after completion of the first one, type errors would be almost impossible to localize. In a practical setting, one runs a complete second phase after each step of the first phase. Thus, type errors can be found as early as possible and be easily localized.

## 6 Conclusion

The relatively complex problem of inferring types for object-oriented languages can be broken up into tractable subproblems, one of which is the record type inference problems.

The type inference systems for a flexible treatment of records resulted from the incapability of existing systems [4, 7] to deal with records of an arbitrary number of labels. Rémy [17, 18] introduced the notion of fields, permitting a treatment of extensible record types using ordinary unification. Wand [25, 26, 24] extended the system of Rémy to deal with infinite label sets and the concatenation operator.

The major achievement of our system compared to Rémy's or Wand's is to get rid of fields and field variables. This makes the formalization much simpler. We get an algorithm that immediately and naturally deals with infinite label sets.

A difference that results from a restriction has to do with classes. Wand wants to admit classes as first-class citizens. Since his record language has concatenation he is unable to infer principal types for expressions. He has to infer sets of principal types. In O'SMALL, classes are not first class citizens. This allows us to calculate all concatenations that are caused by inheritance at compile time. Therefore, our intermediate language **RFI** does not need concatenation. We can infer principal types. In a practical setting, principal types are superior to sets of principal types. Their documentary value is better. If we only consider the record aspects, Wand's language has the following syntax:

$e ::=$	$x$	variable
	$\langle \bar{a} \mapsto \bar{e} \rangle$	record
	$e \oplus e$	concatenation
	$e.a$	selection

The problem is that, in the selection of a concatenation like  $(x \oplus y).a$ , we do not know if the  $a$ -component must be present in  $x$  or in  $y$ . Therefore, principal types cannot be inferred in any of the known systems [26, 17, 4]. The language we have examined in [11] has the following syntax:

$e ::=$	$x$	variable
	$\langle \bar{a} \mapsto \bar{e} \rangle$	record
	$\langle a \mapsto e \rangle \oplus e$	adjunction
	$e.a$	selection

If we extend this language by  $\lambda$ -abstraction and function application, we can formulate the counterexample for principal types [23]. The symbol  $+$  stands for integer addition.

$$\lambda f. \lambda x. f(\langle a \mapsto 3 \rangle \oplus x) + f(\langle a \mapsto 3 \rangle \oplus \langle \rangle)$$

The adjunction of  $\langle a \mapsto 3 \rangle$  to  $x$  must yield a term that has just an  $a$ -field. Thus,  $x$  must either be the empty record or a record with just an  $a$ -field. The types of this term are

$$(\langle a : \text{int} \rangle \rightarrow \text{int}) \rightarrow \langle \rangle \rightarrow \text{int}$$

and

$$(\langle a : \text{int} \rangle \rightarrow \text{int}) \rightarrow \langle a : \tau \rangle \rightarrow \text{int}$$

for any type  $\tau$ . In the present framework, there is no type scheme that can generate just these types.<sup>4</sup>

In order to obtain principal types, we have to simplify things even further. Adjunction has to be limited to records that are “known in advance”. If records are known in advance we can return to general concatenation:

$e$	::=	$x$	variable
		$e.a$	selection
		$r$	simple record
$r$	::=	$\langle \bar{a} \mapsto \bar{e} \rangle$	record
		$r \oplus r$	concatenation

From this language, we obtain **R** by letting the compiler resolve all concatenations. This makes the formalization easier.

The type reconstruction algorithm is divided into two phases, which can be verified separately. This description is similar to that of Wand [22]. The two-phase algorithm is more declarative and easier to verify than the classic algorithm à la Damas, Milner [9]. The resolution phase combines extensible record unification (*padding* [26]) and rational tree unification [8, 13]. The phases are orthogonal and can be applied in any order. An interleaving order in which the resolution phase is run after each step of the extraction phase yields an algorithm similar to the classic one. It detects type clashes as soon as possible.

**Acknowledgements** We thank Martin Müller for comments on a draft version.

## References

- [1] R. Amadio and L. Cardelli. Subtyping recursive types. In *Symposium on Principles of Programming Languages*, pages 104–118, Orlando, Florida, Jan. 1991. ACM.

<sup>4</sup>The bug in [11] is in the adjunction axiom of the typing relation: the term  $\rho' \cdot a[\alpha]$  may not be well-sorted, however, well-sortedness is assumed throughout the paper.

- [2] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance and explicit coercion. In *Logic in Computer Science*, pages 112–128, 1989.
- [3] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded quantification for object-oriented programming. In *Conference on Functional Programming and Computer Architecture*, pages 273–280. ACM, 1989.
- [4] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.
- [5] L. Cardelli. Structural subtyping and the notion of power type. In *Symposium on Principles of Programming Languages*, pages 70–79. ACM, Jan. 1988.
- [6] L. Cardelli and J. C. Mitchell. Operations on records. *Lecture Notes in Computer Science*, 389:75–81, 1989. extended abstract.
- [7] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, Dec. 1985.
- [8] A. Colmerauer. *Logic Programming*, chapter Prolog and Infinite Trees, pages 231–251. Academic Press, 1982.
- [9] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Symposium on Principles of Programming Languages*, pages 207–212. ACM, 1982.
- [10] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Commun. ACM*, 22:465–476, 1979.
- [11] A. V. Hense and G. Smolka. A verification of extensible record types. In Z. Shi, editor, *Proceedings of the IFIP TC12/WG12.3 International Workshop on Automated Reasoning*, pages 137–164, Beijing, P.R. China, 13–16 July 1992. International Federation for Information Processing, Elsevier, North-Holland, Excerpta Medica.
- [12] M. Höhfeld and G. Smolka. Definite relations over constraint languages. LILOG-Report 53, IBM Deutschland, Oct. 1988. to appear in *Journal of Logic Programming*.
- [13] G. Huet. *Résolution d'équations dans les langages d'ordre 1,2,..., $\omega$* . PhD thesis, Université Paris 7, 1976. Thèse de doctorat d'état.
- [14] L. A. Jategaonkar and J. C. Mitchell. ML with extended pattern matching and subtypes. In *Symposium on Lisp and Functional Programming*, 1988.
- [15] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.
- [16] M. J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Logic in Computer Science*, Edinburgh, 1988.

- [17] D. Rémy. Typechecking records and variants in a natural extension of ML. In *Symposium on Principles of Programming Languages*, pages 77–88. ACM, 1989.
- [18] D. Rémy. *Algèbres Touffues. Application au Typage Polymorphe des Objets Enregistrements dans les Languages Fonctionnels*. PhD thesis, Université Paris 7, Feb. 1990.
- [19] G. Smolka, W. Nutt, J. Goguen, and J. Meseguer. *Order-Sorted Equational Computation*, volume 2 of *Resolution of Equations in Algebraic Structures*, chapter 10, pages 297–367. Academic Press, 1989.
- [20] R. Stansifer. Type inference with subtypes. In *Symposium on Principles of Programming Languages*, pages 88–97. ACM, Jan. 1988.
- [21] M. Wand. Complete type inference for simple objects. In *Logic in Computer Science*, pages 37–44, 1987.
- [22] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae X*, pages 115–122, 1987.
- [23] M. Wand. Corrigendum: Complete type inference for simple objects. In *Logic in Computer Science*, page 132, 1988.
- [24] M. Wand. Type inference for objects with instance variables and inheritance. Technical Report NU-CCS-89-2, Northeastern University, Boston, 1989.
- [25] M. Wand. Type inference for record concatenation and multiple inheritance. In *Logic in Computer Science*, pages 92–97, 1989.
- [26] M. Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, July 1991.